TABC: THE COMPILER OF THE TABLA FRAMEWORK

A Thesis
Presented to
The Academic Faculty

by

Joon Kyung Kim

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science in the
College of Computing

Georgia Institute of Technology
May 2016

TABC: THE COMPILER OF THE TABLA FRAMEWORK

Approved by:

Professor Hadi Esmaeilzadeh, Advisor
College of Computing
Georgia Institute of Technology

Professor Tushar Krishna
College of Computing
Georgia Institute of Technology

Date Approved: 3 May 2016

To everyone

in the Alternative Computing Technologies Laboratory,

ACKNOWLEDGEMENTS

I want to thank Chenkai Shao and Joonho Kim for their contributions. Chenkai is an undergraduate student in the School of Electrical and Computer Engineering. Joonho is an undergraduate student in College of Computing. This work would not have been possible without their commitment and dedication.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

TABLA is an innovative framework that accelerates statistical machine learning algorithms. Given the programmer's code, TABLA generates a hardware accelerator, in the form of synthesizable Verilog code. It contains a set of template hardware components pre-designed by expert hardware designers. Given the input code, TABLA generates multiples of this component in order to maximize parallelism. By automating this process, TABLA alleviates the arduous task of custom hardware design from software engineers. TABLA-generated acceleraters show significant speedup over commercially available CPU and GPU platforms.

TABLA provides a new domain specific language. It is designed to make implementation of statistical machine learnign algorithms easier. The compiler's primary job is to schedule the order of operations that yields the most optimal runtime performance. The first step is a data dependency analysis. This information is stored in the form of dataflow graph. It consists of nodes and edges that represent mathematical operations and dependencies, respectively. Dataflow graph is essential information needed by the scheduling algorithm. Optimal scheduling is crucial to improving the perfornamce of the target algorithms. The primary purpose of this paper is to describe the design and implementation of the compiler in detail.

CHAPTER I

INTRODUCTION

A popular trend in computing industry is application of machine learning algorithms. As the name suggests, machine learning trains computers learn the pattern in data. This is of significant importance to many applications, because it allows computers to make useful predictions of unforseen data. Examples of machine learning applications include language detection, handwriting recognition, and weather prediction.

This revolutionary paradigm makes heavy use of big data. However, there is a growing gap between the demand for more data and the capabilities of hardware platforms, as shown on the Dark Silicon study [2]. While the demand for big data is increasing, most hardware platforms cannot meet the expectations to deliver necessary performance results. As a matter of fact, the performance benefits promised by Moore's Law is projected by many to come to an end. TABLA is an effort to find alternative solutions to match the growing needs of machine learning algorithms.

Field Programmable Gate Arrays (FPGAs) have been gaining popularity among hardware designers as a platform of choice, thanks to its programmability and efficiency. As the name suggests, FPGAs are used to design custom hardware tailored to an application. The difference between general purpose cores and FPGAs is that the former contains pre-fabricated architecture, whereas the latter can be programmed to contain only the hardware resources needed by the application. Moreover, FPGAs offer more generality than Application Specific Integrated Circuits (ASICs). ASICs are target a specific application, whereas FPGAs can be reconfigured. To summarize, FPGAs provide both the flexibility of general-purpose cores and efficiency of ASICs.

However, there is a big hurdle to this approach; programming FPGAs requires extensive knowledge and experience in hardware design. It is impractical to expect high level language programmers to design optimal hardware for their needs. Therefore, our solution, TABLA, provides a comprehensive framework that takes programmers code as input and generates synthesizable Verilog code, which generates the hardware accelerator. This allows programmers to focus on implementing machine learning algorithms without having to worry about the hardware details. Our work abstracts away the details of hardware design from programmers, while providing them with an environment to stay high level and achieve maximum performance benefits.

CHAPTER II

LITERATURE REVIEW

There are several examples of using FPGAs for accelerating a domain of applications. Putnam et al. successfully integrated 1632 FPGAs in Microsoft BIng servers to accelerate their search ranking algorithm [5]. The work by Chung et.al implemented database query acceleration using FGPAs [1].

There have been several studies on accelerating machine learning algorithms. However, their domain of ML algorithms tend to be limited to a particular machine learning task. For example, the work by Yeh et al. focuses on accelerating k-NN classifier algorithm [6]. Some of the research includes accelerating k-NN, k-Means, and support vector machines. The work by Liu et al. developed PuDianNao, an accelerator for accelerating seven machine learning algorithms [4]. However, this does not offer flexibility in acceleration of machine learning algorithms, since this is done on an ASICs.

CHAPTER III

OVERVIEW OF TABLA

TABLA is an accelerator generator. This fundamentally differs from accelerators tailored to optimize a specific application. TABLA is capable of generating custom hardware by analyzing input code. It provides an environment for programmers to implement a class of machine learning algorithms without having to worry about the details of hardware and accelerator design. We focus on a class of algorithms that can be solved by applying stochastic gradient descent algorithms. The following are benchmarks used in the paper to measure the performance of TABLA: logistic regression, classification, linear regression, support vector machines, recommender systems, and backpropagation. All these algorithms have one major commonality: stochastic gradient descent algorithm. TABLA takes full advantage of this aspect. It comes with pre-designed accelerator templates for stochastic gradient descent. This takes weight off the programmers' shoulders; their only job is to implement the objective functions.

TABLA contains multiple components. First, a new domain specific language is provided to programmers. It is designed to simplify implementation of stochastic gradient descent learning algorithms. Because a new language requires a compiler for it, TABLA contains a compiler for the language. The primary purpose of the compiler is to generate the optimal program execution schedule. It takes the programmers code written in the TABLA language as an input. It then performs a data dependency analysis and outputs what is known as dataflow graph. Dataflow graph is an effective representaiton of data dependencies in a given program. This information is passed to the scheduler. Operation scheduler decideds the order of operations to be executed

in the most optimal manner. Efficient scheduling is crucial to achieve maximum performance. This paper mostly describes the compiler in detail.

The next component of TABLA is the design builder. It analyzes programmers' input and generates synthesizable Verilog code for the accelerator. Specifically, design builder takes as input the objective function, a high level specification of the target FPGA, and the predesigned template in Verilog that correspond to the stochastic gradient descent.

CHAPTER IV

THE TABLA LANGUAGE

This chapter explains the language features in detail. It lays out the foundation for the implementations shown in the next chapter.

4.1 Lexical Rules

In TABLA, data can be saved in variables. Variable names follow the similar rules as the ones in the C programming language. They start with either an upper case letter or an upper case letter, followed by an arbitrary length of alphanumeric characters, including an underscore (_). It can also end with a single quote ('). Variables can be of any dimension. For example, the following are all legal:

```
a, b[m], c[x][y], d[i][j][k]
```

The TABLA language supports operator precedence. The operators follow the following precedence, from highest to lowest:

```
(), []
*
+, -
<, >
=
```

4.2 Grammar

Every statement in the TABLA langauge end with a semicolon(;). The TABLA language is not a strongly-typed language. However, it does enforce data types for

initial input data, such as model and iterator. There are five data types supported in TABLA:

```
model_input
model_output
model
gradient
iterator
```

Variables of these data types must be declared. Multiple variables of the same data type can be decalred in the same line, even if they have different dimensions. For example, the following is legal:

```
model_input i[x], j[y][z];
```

On the other hand, intermediate variables do not have to be decalred. For example,

```
m = 10;
```

This is a valid statement, even though the variable m does not have a data type associated with it explicitly. The following code snippet is legal:

```
m = 15; model_input x[m];
```

However, the following is not legal, since n is not declared:

```
model_output y[n];
```

Iterator data type has a special syntax associated with its variables. A variable name is immediately followed by a left bracket, start and end indices delimited by a colon, and a right bracket. In other words,

```
(data type) (variable name)(left bracket)(digit or an
    integer variable)(colon)(digit or an integer variable)(
    right bracket)
```

Using a token notation,

ITERATOR ID LEFT_BRACK (ID | INTLIT) COLON (ID | INTLIT) RIGHT_BRACK SEMI

The iterator data type serves the same functionality as a for loop in most programming languages. The range of values to be looped is expressed inside the brackets. These are integer values in increments of one. Either raw values or variables containing an integer value (or both) can be used for this. Here are examples of valid iterator declartion:

```
iterator i[0:10]; // all iterator arguments as integer
    literals
iterator j[m:n]; // all iterator arguments as integer
    variables (that should havebeen decalred before this
    statment)
iterator k[m:10]; // one iterator argument as an integer
    variable, the other as an integer literal
iterator l[0:n]; // same as before, but the other way
    around
```

The following is illegal, since it does not give the range of values to be looped:

```
iterator x;
```

'-¿' notation is used to associate each gradient variable with corresponding model variable.

4.3 Functions

Aside from the basic operators, there are two types of operations supported by the TABLA language: group and non linear. In group operations, *pi* and *sum* operates on two arguments, whereas norm operates on one argument. However, even though *pi*

and *sum* operates on two arguments, this is only in a semantic manner. Syntatically, it appears they take in one. In other words, in between the parentheses, *pi* and *sum* operators do not require an argument followed by a comma and then another argument, unlike one would normally expect from other languages. For example, if one would write a sum function in C, it would look like:

```
sum(2, 3);
```

However, in TABLA, it would look something like this:

```
sum[i](x[i] * w[j][i]);
```

where i and j are iterators. Notice there is no comma (,) inside the parentheses. Because *pi* and *sum* are operated group-wise, they require an iterator. This is wrapped inside square brackets, as shown above in the *sum* operator. Syntatically, *sum* and *pi* operators come in the following format:

```
(SUM | PI) LEFT_BRACK ID RIGHT_BRACK LEFT_PAREN expr
    RIGHT_PAREN SEMI
```

whereas the rest of the operators are expressed in the following format:

```
(NORM | GAUSSIAN | SIGMOID | SIG_SYM | LOG) LEFT_PAREN expr
    RIGHT_PAREN SEMI
```

4.4 Examples

Here is an example implementation of linear regression algorithm using the TABLA language.

As shown, the implementation can be done in less than 15 lines of code. The last three commented lines are shown on purpose, in order to illustrate that the process of stochastic gradient descent is already handled internally by the TABLA framework. There is no need for programmers to implement this portion.

```
mu = 1;
m = 3; // num of features

model_input x[m]; // Assume x[0] is 1
model_output y;
model w[m];
gradient g[m] -> w;

iterator i[0:m];

h = sum[i](w[i] * x[i]);
d = h - y;
g[i] = d * x[i];

// SGD added
//g[i] = mu * g[i];
//w[i] = w[i] - g[i];
```

Figure 1: Implementation of linear regression using the TALBA language

CHAPTER V

DESIGN AND IMPLEMENTATION OF THE COMPILER

This section gives a detailed explanation on the design and implementation of the compiler. Most of the funcionalities are implemented in Python version 3.4.3. Details of other tools and libraries used will be described in the following sections in this chapter.

5.1 Front end: Lexical Scanning and Parsing

5.1.1 Design

The first step in any compilation process is lexical scanning and parsing. The full specifications of the language have been provided in the previous section. This section lists the third party libraries used to generate the parser. The grammar has been devised exclusively by the TABLA group. We used the ANTLR parser generator (version 4) to automatically generate the parser. ANTLR provides Python target code, even though its popular use is Java. Using the parser, any input code conforming to the grammar rules of the TABLA language is used to generate a parse tree, a fundamental piece of information used by the later processes in the compiler.

5.1.2 Implementation

Lexical tokens are specified as regular expressions. Then, ANTLR accepts grammar expressed in LALR(*) formats. Both tokens and grammar can be specified in the same file. This simplifies the generation of parsers by ANTLR.

Here is a complete specification of the lexical rules.

```
grammar Tabla;
```

```
/* scanner tokens */
MODEL_INPUT : 'model_input';
MODEL_OUTPUT : 'model_output';
MODEL : 'model';
GRADIENT : 'gradient';
ITERATOR : 'iterator';
ADD : '+';
SUB : '-';
LT : '<';
GT : '>';
MUL : '*';
PI : 'pi';
SUM : 'sum';
NORM : 'norm';
GAUSSIAN : 'gaussian';
SIGMOID : 'sigmoid';
SIG_SYM : 'sigmoid_symmetric';
LOG : 'log';
SEMI : ';';
COLON: ':';
LEFT_BRACK : '[';
RIGHT_BRACK : ']';
LEFT_PAREN : '(';
RIGHT_PAREN : ')';
COMMA : ',';
ASSIGN : '=';
```

```
/* var name */
ID
    : (LOWER | UPPER) (LOWER | UPPER | DIGIT | '_')* ('\'')?
    ;
fragment LOWER: 'a'..'z';
fragment UPPER: 'A'..'Z';
fragment DIGIT: '0'..'9';
WHITESPACE
    : (' ' | '\t' | '\n' | '\r')+ -> skip
    ;
COMMENT
//    : '/*' .*? '*/'
    : '//' .+? ('\n' | EOF) -> skip
    ;
INTLIT
    : '0'
    | '1'..'9' (DIGIT)*
    ;
```

Here is a complete expression of the grammar.

```
program
    : data_decl_list stat_list EOF
    ;
data_decl_list
    : data_decl*
    ;
data_decl
```

```
    : data_type SEMI
    ;
data_type
    : non_iterator var_list
    | GRADIENT var_with_link_list
    | iterator var_list_iterator
    | ID ASSIGN INTLIT
    ;
non_iterator
    : MODEL_INPUT
    | MODEL_OUTPUT
    | MODEL
//    | GRADIENT
    ;
iterator
    : ITERATOR
    ;
var_with_link_list
    : var '->' var var_with_link_list_tail
    ;
var_with_link_list_tail
    : ',' var_with_link_list var_with_link_list_tail
    | // elipse
    ;
var_list
    : var var_list_tail
    ;
```

```
var
    : var_id
    ;
var_id
    : ID id_tail
    ;
id_tail
    : LEFT_BRACK (ID | INTLIT) RIGHT_BRACK id_tail
        | // epsilon
    ;
var_list_tail
    : COMMA var_list
    | // epsilon
    ;
var_list_iterator
    : ID LEFT_BRACK (ID | INTLIT) COLON (ID | INTLIT)
      RIGHT_BRACK
    ;
stat_list
    : stat*
        | // epsilon
    ;
stat
    : var ASSIGN expr SEMI
    ;
expr
    : term2 term2_tail
```

```
    ;
function
    : PI
        | SUM
        | NORM
        | GAUSSIAN
        | SIGMOID
        | SIG_SYM
        | LOG
    ;
function_args
    : LEFT_BRACK ID RIGHT_BRACK LEFT_PAREN expr RIGHT_PAREN
    | LEFT_PAREN expr RIGHT_PAREN
    ;
term2_tail
    : compare_op term2 term2_tail
        | // epsilon
    ;
term2
    : term1 term1_tail
    ;
term1_tail
    : add_op term1 term1_tail
        | // epsilon
    ;
term1
    : term0 term0_tail
```

```
    ;
term0_tail
    : mul_op term0 term0_tail
        | // epsilon
    ;
term0
    : var
    | LEFT_PAREN expr RIGHT_PAREN
    | INTLIT
    | function function_args
    ;
mul_op
    : MUL
    ;
add_op
    : ADD
    | SUB
    ;
compare_op
    : LT
    | GT
    ;
```

These two parts are expressed in one file used by ANTLR.

5.2 Back end: Generation of Dataflow Graph

5.2.1 Design

Dataflow graph is an effective representation of mathematical operations and its dependencies. Specifically, it is a directed acyclic graph starting from source node and ending with sink node. Source and sink nodes merely designate start and end of operations, respectively. Dataflow graph is generated by walking the parse tree It consists of nodes and edges that represent mathematical operations and dependencies, respectively. The immediate children nodes of the source node represent inital data used in the learning algorithm, so that users can easily identify the initial values before any operations are performed. Subsequent children nodes represent the operations and functions specified in the input code. For example, figure 2 corresponds to the linear regression example from the previous chapter.

We represent the data type of nodes by coloring them. Blue nodes are model_input, yellow is model, pink is model_output, gray is constant, and green is gradient. The intermediate nodes are left without any color, because it is implausible to confine them to one particular data type. For example, it does not make sense to assign data type to the product of model_input andmodel data.

As shown in 2, dataflow graph is a very effective graphical representation of data dependencies in programmer-specified code. This is essential information needed by the scheduling algorithm.

5.2.2 Implementation

The first step in dataflow graph generation is walking the parse tree. Starting from the root node, the parse tree is traversed all the way down to leaf nodes. Once the leaf node is reached, the information regarding operations and data is retrieved and returned back to the calling function [2]. This data returns up to the statement level. For every statement in the given program, the tree traversal function determines the

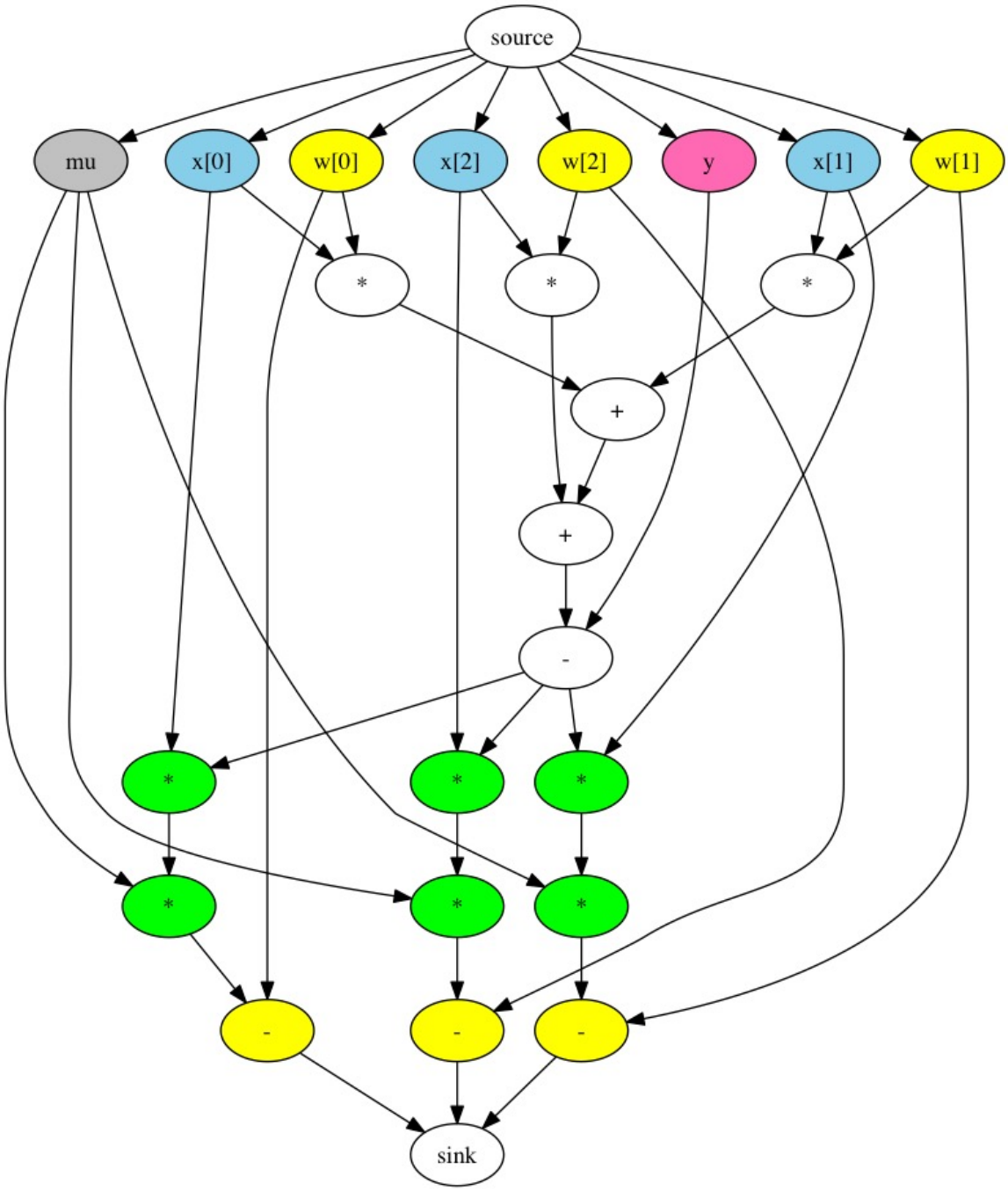

Figure 2: Dataflow graph of linear regression

```
{
  "children": [
    2,
    3,
    4,
    5,
    6,
    7,
    8,
    9
  ],
  "dataType": null,
  "dist2sink": 9,
  "id": 0,
  "operation": "source",
  "parents": []
}
```


Figure 3: Example dataflow graph node represented in JSON format

information on the type of input data, operation, parent and children nodes, and node id. The final output of dataflow graph generation is in JSON format. Here is an example of a node represented in JSON format. This is the source node from the linear regression code shown in the previous chapter 4.4 One attribute of particular importance is distance to sink (dist2sink). As the name suggests, this is the number of edges between the current node and the sink node. This is critical information needed by the scheduling algorithm, described in detail in the following section.

5.3 Operation Scheduling

5.3.1 Design

Using data flow graph as input, Tabla executes ML-RCS (Minimum Latency-Resource Constrained Scheduling) alrogithm to effectively schedule the opertions, given the hardware resource constraints [3]. Also known as Hu's Algorithm, ML-RCS outputs operations to be performed cycle-by-cycle. All of the graphical representations of

data flow graphs reflect the scheduled operations; each horizontal layer corresponds to each cycle, excluding the source, sink, and initial data nodes. The algorithm is shown below.

```
Require: R: Set of available resources
         O: Set of all the operations to be scheduled
         D: Distance to sink for each operation
Ensure: S: Final schedule
   Initialize S ← ∅
   Initialize current_cycle ← 0
   while (O ≠ ∅) do
      for (r ∈ R) do
         if o ∈ O where o.predecessors = DONE & o.distance = max(D) then
            schedule.op = o; schedule.resource = r; schedule.cycle = current_cycle
            S.append(s)
            O.remove(o)
         end if
      end for
      current_cycle = current_cycle + 1
   end while
```

Algorithm 1: **Minimum-latency resource constrained scheduling.**

5.3.2 Implementation

The compiler directly implements the ML-RCS algorithm shown above. An essential piece of information is distance from the sink node. The bigger the value, the higher the priority. Another aspect considered is the avilability of resources. The scheduler takes the number of Processing Engines as its input. When run out of resources, it can schedule operations to be performaed on the following cycle, even if it has the same prioirity as other scheduled processes. The graphical representation of dataflow graph reflects the schedule as well. The nodes are horizontally aligned according to their scheduled cycle. For example, three multiplications will be done in the first cycle in the figure. Then one addition will be done in the next cycle, and so on.

5.4 Visualization of Dataflow Graph

5.4.1 Design

Graphical representation of dataflow graph is a very effective way of visualizing the operations scheduled cycle-by-cycle. We use a graph visualization tool on each benchmark dataflow graph, in order to check the validity of the scheduler.

5.4.2 Implementation

The GraphViz tool is used to visualize dataflow graphs. After scheduling, a dot file is generated by parsing the dataflow graph JSON file. Here is a dot file generated for the dataflow graph of linear regression.

```
digraph G {
"source" -> {"2" [label="mu" style=filled fillcolor="gray"]};
"source" -> {"3" [label="x[0]" style=filled fillcolor="
    skyblue"]};
"source" -> {"4" [label="x[1]" style=filled fillcolor="
    skyblue"]};
"source" -> {"5" [label="x[2]" style=filled fillcolor="
    skyblue"]};
"source" -> {"6" [label="y" style=filled fillcolor="hotpink
    "]};
"source" -> {"7" [label="w[0]" style=filled fillcolor="yellow
    "]};
"source" -> {"8" [label="w[1]" style=filled fillcolor="yellow
    "]};
"source" -> {"9" [label="w[2]" style=filled fillcolor="yellow
    "]};
```

```
{"2" [label="mu" style=filled fillcolor="gray"]} -> {"19" [
   label="*" style=filled fillcolor="green"]};
{"2" [label="mu" style=filled fillcolor="gray"]} -> {"21" [
   label="*" style=filled fillcolor="green"]};
{"2" [label="mu" style=filled fillcolor="gray"]} -> {"23" [
   label="*" style=filled fillcolor="green"]};
{"3" [label="x[0]" style=filled fillcolor="skyblue"]} ->
   {"10" [label="*"]};
{"3" [label="x[0]" style=filled fillcolor="skyblue"]} ->
   {"16" [label="*" style=filled fillcolor="green"]};
{"4" [label="x[1]" style=filled fillcolor="skyblue"]} ->
   {"11" [label="*"]};
{"4" [label="x[1]" style=filled fillcolor="skyblue"]} ->
   {"17" [label="*" style=filled fillcolor="green"]};
{"5" [label="x[2]" style=filled fillcolor="skyblue"]} ->
   {"12" [label="*"]};
{"5" [label="x[2]" style=filled fillcolor="skyblue"]} ->
   {"18" [label="*" style=filled fillcolor="green"]};
{"6" [label="y" style=filled fillcolor="hotpink"]} -> {"15" [
   label="-"]};
{"7" [label="w[0]" style=filled fillcolor="yellow"]} -> {"10"
    [label="*"]};
{"7" [label="w[0]" style=filled fillcolor="yellow"]} -> {"20"
    [label="-" style=filled fillcolor="yellow"]};
{"8" [label="w[1]" style=filled fillcolor="yellow"]} -> {"11"
    [label="*"]};
```

```
{"8" [label="w[1]" style=filled fillcolor="yellow"]} -> {"22"
    [label="-" style=filled fillcolor="yellow"]};
{"9" [label="w[2]" style=filled fillcolor="yellow"]} -> {"12"
    [label="*"]};
{"9" [label="w[2]" style=filled fillcolor="yellow"]} -> {"24"
    [label="-" style=filled fillcolor="yellow"]};
{"19" [label="*" style=filled fillcolor="green"]} -> {"20" [
   label="-" style=filled fillcolor="yellow"]};
{"21" [label="*" style=filled fillcolor="green"]} -> {"22" [
   label="-" style=filled fillcolor="yellow"]};
{"23" [label="*" style=filled fillcolor="green"]} -> {"24" [
   label="-" style=filled fillcolor="yellow"]};
{"10" [label="*"]} -> {"13" [label="+"]};
{"16" [label="*" style=filled fillcolor="green"]} -> {"19" [
   label="*" style=filled fillcolor="green"]};
{"11" [label="*"]} -> {"13" [label="+"]};
{"17" [label="*" style=filled fillcolor="green"]} -> {"21" [
   label="*" style=filled fillcolor="green"]};
{"12" [label="*"]} -> {"14" [label="+"]};
{"18" [label="*" style=filled fillcolor="green"]} -> {"23" [
   label="*" style=filled fillcolor="green"]};
{"15" [label="-"]} -> {"16" [label="*" style=filled fillcolor
   ="green"]};
{"15" [label="-"]} -> {"17" [label="*" style=filled fillcolor
   ="green"]};
{"15" [label="-"]} -> {"18" [label="*" style=filled fillcolor
   ="green"]};
```

```
{"20" [label="-" style=filled fillcolor="yellow"]} -> "sink";
{"22" [label="-" style=filled fillcolor="yellow"]} -> "sink";
{"24" [label="-" style=filled fillcolor="yellow"]} -> "sink";
{"13" [label="+"]} -> {"14" [label="+"]};
{"14" [label="+"]} -> {"15" [label="-"]};
{rank = source; "source";};
{rank = same; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9"; };
{rank = same; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9"; };
{rank = same; "11"; "10"; "12"; };
{rank = same; "13"; };
{rank = same; "14"; };
{rank = same; "15"; };
{rank = same; "18"; "17"; "16"; };
{rank = same; "23"; "21"; "19"; };
{rank = same; "24"; "22"; "20"; };
{rank = sink; "sink";};
}
```

CHAPTER VI

CONCLUSION

TABLA-generated accelerators show significant performance improvements over CPU and GPU platforms, despite its deficiencies over some high-tier GPU platforms such as Nvidia Tesla K40. TABLA is a comprehensive framework. Its solution spans from a high-level programming language to synthesizable Verilog code. Thanks to its novelty and significance, the TABLA paper has been awarded the Distinguised Paper Award from the High Performance Computer Architecture conference in March 2016, a very prestigious and selective award.

This paper provides more in-depth explanation of the compiler presented in the original TABLA paper, focusing on the design and implementation. Data dependency analysis and operation scheduling is crucial to achieving the performance improvements promised by TABLA. Moreover, TABLA provides an easy-to-learn domain specific langauge. The only task left for programmers is to implement the objective function of the statistical learning algorithm of their choice. The beauty of TABLA is that the programmers need not worry about the underlying details of hardware accelerator design.

We hope TABLA gets a widespread use in both academia and industry.

REFERENCES

[1] Chung, E. S., Davis, J. D., and Lee, J., "LINQits: Big data on little clients," in *ISCA*, 2013.

[2] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D., "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[3] Ku, D. C. and De Micheli, G., *High level synthesis of ASICs under timing and synchronization constraints*. Kluwer Academic Publishers, 1992.

[4] Liu, D., Chen, T., Liu, S., Zhou, J., Zhou, S., Teman, O., Feng, X., Zhou, X., and Chen, Y., "PuDianNao: A polyvalent machine learning accelerator," in *ASPLOS*, 2015.

[5] Putnam, A., Caulfield, A., Chung, E., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Prashanth, G., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J. R., Peterson, E., Smith, A., Thong, J., Xiao, P. Y., and Burger, D., "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, June 2014.

[6] Yeh, Y.-J., Li, H.-Y., Hwang, W.-J., and Fang, C.-Y., "FPGA implementation of kNN classifier based on wavelet transform and partial distance search," in *SCIA*, 2007.